Introduction to OCaml

Mattox Beckman

beckman@iit.edu

Illinois Institute of Technology

Your goal in today's lecture is to gain some familiarity with OCaml. In particular, you should know. . .

- how to use immediate mode for interactive programming.
- the basic builtin types.
- how to use pattern matching in your functions.
- how to use tuples and lists.
- know the four ways to create variables, and how long they last.
- that **whenever you see a function, you always ask "what's its type?".**

Language Features:

- higher order functional language
- call by value parameter passing style
- modern syntax
- parametric polymorphism
- automatic garbage collection

And two more things....

- It's very fast — the winners of the 2004, 2000 and 1999 ICFP Programming Contests used OCaml.
- The error messages make sense.

- Immediate Mode statements must be terminated by a `;;`
- Notice the type inference!

```
1 $ ocaml
2         Objective Caml version 3.04
3
4 # 34 + 8;;
5 - : int = 42
6 # 27.0 +. 9.4;;
7 - : float = 36.4
8 # "hello";;
9 - : string = "hello"
```

- This is the Standard First Program™
- The type `unit` is like `void` in C/C++; it represents *commands*.

```
# print_string "Hello, world!\n";;
Hello, world!
- : unit = ()
```

- The type `unit` has one value, `()` (pronounced “unit” or “dot”).
- Warning: “output” has two meanings:
  - “Hello, world!” has been output (I/O) to the screen, *but*
  - the output (result) of this function is `unit`.
  - In CS 440 output will always refer to *result*.


```
#  20.3;;
- : float = 20.3
# 42;;
- : int = 42
# (answer = 42);;
- : bool = true
# "Ravi";;
- : string = "Ravi"
#
```


- If you try to combine things with incompatible types, you get a *type error*.
- You really need to learn how to read these.

```
# 1 + "hello";;
      ^^^^^^^
This expression has type string but is here used
     with type int
# 2.0 +. 4;;
         ^
This expression has type int but is here used
     with type float
#
```


- Variables behave differently in a functional language than in an imperative language!
- Variables are created, and then never changed.
- Modifications are done by *copying*.
- Very important: be able to know when a variable is created and when it is destroyed.

Variable Creation Method 1: Global Let.

```
# let answer=42;;
val answer : int = 42
# answer * 6;;
- : int = 252
```

answer | 42 |

Variable Creation Method 2: Function Parameter

`let` *name parameters* = *body*

```
# let inc x = x + 1;;
val inc : int -> int = <fun>
# inc 5;;  (* Notice the arrow! *)
- : int = 6
# inc 2.4;;
Toplevel input:
# inc 2.4;;
      ^^^
This expression has type float but is here used
     with type int
```

Parameters exist *only within the function*.

Memory trace...

```
# let inc x = x + 1;;
# inc 5;;
# inc 2;;
# inc 6;;
```

inc –stuff–
x 5
x 2
x 6

Each value of `x` is created when `inc` is called, and destroyed when `inc` returns.

Functions with Multiple Parameters

Functions are *values* in this language!

```
# let triangle a b = a * a + b * b;;
val triangle : int -> int -> int = <fun>
# triangle 3 4;;
- : int = 25
# let t3 = triangle 3;;
val t3 : int -> int = <fun>
# t3 4;;
- : int = 25
# let inc = fun x -> x + 1
val inc : int -> int = <fun>
# inc 3;;
- : int = 4
```


Variable Creation Method 3: Local Let

- You can create local variables using the `let/in` construct.

```
# let triangle a b =
     let asq = a * a in
     let bsq = b * b in
       asq + bsq;;
val triangle : int -> int -> int = <fun>
```

- The variables `asq` and `bsq` are created after `triangle` is called, once the `let` expressions are reached, and destroyed once the `let` expressions are exited.

```
# (let x = 10 in x) + (let x = 20 in x);;
- : int = 30
```

Memory Trace

```
# let triangle a b =
     let asq = a * a in
     let bsq = b * b in
       asq + bsq;;
val triangle : int -> int -> int = <fun>
```

triangle –stuff–

a 3

b 4

asq 9

bsq 16

Scoping Question

Consider this code: (This is question #15 in the database.)

```
let x = 27;;
let foo x =
   let x = 5 in
     (fun x -> print_int x) 10;;
foo 12;;
```

What value will be printed?

- a) 5
- b) 10
- c) 12
- d) 27

Tuples

- Tuples are one kind of *compound type*.

```
# let t = 2,3;;
val t : int * int = 2, 3
# let t2 = (3,42);;
val t2 : int * int = 3, 42
# let ultimate = "answer",42;;
val ultimate : string * int = "answer", 42
# let big = (2,3,"hi",4.123,"there",triangle);;
val big : int * int * string * float *
          string * (int -> int -> int) =
  2, 3, "hi", 4.123, "there", <fun>
#
```


Pattern Matching

```
# let iszero n =
     match n with
       0 -> "it's zero"
     | 1 -> "it's one"
     | _ -> "not zero or one";;
val iszero : int -> string = <fun>
# iszero 1;;
- : string = "it's one"
```

- The *patterns* all need to have the same type.
- Also, the results....

Variable Creation Method 4: Matching

- Use `match`/`with` to deconstruct compound types,

```
# let inctup a =
    match a with
      (0,y) -> y, 1
    | (x,y) -> x+1, y+1;;
val inctup : int * int -> int * int = <fun>
# inctup (2,3);;
- : int * int = 3, 4
# inctup (0,3);;
- : int * int = 3, 1
```

A variable created with `match` lasts only in the expression after the `->`.

Lists

- A *list* can take two forms:
  - It can be an empty list, or
  - it can be an element, together with another list.
- Empty lists are written `[]`
- Non-empty lists are written x`::`xs
  - x is the *head* of the list.
  - xs is the *tail* of the list.
- You can also write them as $[x_1; x_2; \ldots; x_n]$
- Unlike tuples, lists are *monomorphic*—all the elements must have the same type.

List Examples

```
# [];;
- : 'a list = []
# let empty = [];;
val empty : 'a list = []
# let single = [1];;
val single : int list = [1]
# let rlist = [2.3; 4.2; 5.3];;
val rlist : float list = [2.3; 4.2; 5.3]
# let badlist = [3; 4; 3.14159];;
Characters 21-28:
This expression has type float but is here used
     with type int
```


Adding to lists

- `::` adds an element to a list; `@` appends two lists together.

```
# let l1 = [3;6;9];;
val l1 : int list = [3; 6; 9]
# let l2 = [4;7;10];;
val l2 : int list = [4; 7; 10]
# 5 :: l1;;
- : int list = [5; 3; 6; 9]
# 10 :: 20 :: l2;;
- : int list = [10; 20; 4; 7; 10]
# l1 @ l2;;
- : int list = [3; 6; 9; 4; 7; 10]
```

Pattern Matching with Lists

- The `match`/`with` construction works with lists, too.

```
# let getfirst l =
    match l with
      [] -> 0
    | x::xs -> x;;
val getfirst : int list -> int = <fun>
# getfirst [];;
- : int = 0
# getfirst [3;4;5];;
- : int = 3
```


Polymorphic List Functions

```
# let isempty l =
    match l with
      [] -> "yes"
    | _  -> "no";;
val isempty : 'a list -> string = <fun>
# isempty ["hi";"there"];;
- : string = "no"
# isempty [2;3];;
- : string = "no"
```


Combining Types

- Types can be combined arbitrarily
- "Orthogonality principle" — different features don't interfere with each other.

```
# let e1 = [ [20;30]; [10;5;2]; [3;6]; []];;
val e : int list list = [[20; 30]; [10; 5; 2]; ...
# let e2 = [ 2,3; 4,5; 6,7 ];;
val e2 : (int * int) list = [2, 3; 4, 5; 6, 7]
```


Other syntax: if

- The `if` construct is both a *command* and an *expression*.
- The `then` and `else` branches must have the same type.

```
# if (3 < 5) then print_string "hi!\n";;
hi!
- : unit = ()
# let x = 10;;
val x : int = 10
# let y = (if x < 10 then 40 else 50) * 2;;
val y : int = 100
#
```

Problems 1–2

1. What will be the output of the following code?

```
1 let x = 20;;
2 let f = fun x -> x + 1;;
3 let y = f 30;;
4 print_int x;;
```

2. One of the lists below is invalid. Which one?
   - a) `[2; 3; 4; 6]`
   - b) `[2,3; 4,5; 6,7]`
   - c) `[2.3,4; 3.2,5; 6,7.2]`
   - d) `[["hi"; "there"]; ["how"]; []; ["goezit"]]`

Answer 1

What will be the output of the following code?

```
1 let x = 20;;
2 let f = fun x -> x + 1;;
3 let y = f 30;;
4 print_int x;;
```

Answer: 20

Answer 2

One of the lists below is invalid. Which one?

- a) `[2; 3; 4; 6]`
- b) `[2,3; 4,5; 6,7]`
- c) **Answer** `[2.3,4; 3.2,5; 6,7.2]`
- d) `[["hi"; "there"]; ["how"]; []; ["goezit"]]`

The first two elements are of type `float * int`, but the last element is of type `int * float`.

Problems 3–5

What is the type of the following functions?

```
1 let f x = x + 1
2 let g x = x :: [1]
```

Write an OCaml function that inspects a list. If the list is empty, output `"empty"`, otherwise, output `"not empty"`.

Answers 3–4

What is the type of the following functions?

```
1 let f x = x + 1
2 let g x = x :: [1]
```

```
1 f : int -> int
2 g : int -> int list
```


Answer 5

Write an OCaml function that inspects a list. If the list is empty, output `"empty"`, otherwise, output `"not empty"`.

```
1 let isEmpty x =
2   match x with
3   | [] -> "empty"
4   | _ -> "not empty"
```

Next time: The second most powerful idea in Computer Science.